The basic solution (what I implemented in the code):

1. Load the pdf and parse it:
    a. Open PDF: Load the PDF and extract text and word-level metadata (e.g., position) from each page.
    b. Track Word Positions: Identify and map the position of the first word in each line.
    c. Extract the line metadata from each first word metadata
    d. Remove Header and Footer: Exclude text located in the header and footer regions based on vertical position.
    e. Format Text: Adjust punctuation, reverse misaligned words, and ensure proper formatting for English and Hebrew text.
    f. Clean and Return Text: After processing all pages, return the cleaned and formatted text, excluding headers/footers.
2. Create RAG
    a. Split text into chunks using RecursiveCharacterTextSplitter (langchain)
    b. Create rag from chunks using FAISS (since its a single short document - rag DB is in-memory, i,e faiss can be used)
    c. RAG uses hebrew embedding model: “sentence-transformers-alephbert” (through SentenceTransformer)
    d. The embedding are normalized 0-1 to be able to threshold irrelevant chunks (thresh=0.2)
    e. Only k=5 most relevant chunks are returned if they are with similarity higher than threshold
3. Query the LLM model
    a. Create an LLM prompt using the user query and the retrieved chunks from RAG
    b. The context is inserted into a prompt template with Chain of thought guidelines
    c. The LLM model used is “dictalm2.0-instruct:f16” (unfortunately my pc is without GPU so I used the best small hebrew llm from hebrew leaderboard)
    d. The LLM is prompted using OLLAMA API
4. Metrics - bert similarity
    a. I used hebrew bert model “xlm-roberta-base” for calculations
    b. Using transformers and bert_score libraries I calculated the bertscore Precision, Recall, F1 and Similarity
    c. I created validation dataset with reference answers (correct and wrong reference answers)
    d. I tested using 2 ways:
        i. On the questions above: Similarity between LLM response and user query (checked Recall, F1 and Similarity values)
        ii. Validation dataset: Comparison between LLM response and predefined answers (checked Precision, F1 and Similarity values)
            1. Compared the values to correct and wrong answers

High-Level Cloud Architecture Overview

The system will be comprised of multiple interconnected components that facilitate document ingestion, processing, analysis, and summarization. It will include services for data reception and response (via API Gateway), document parsing and cleaning, natural language processing (NLP), text chunking and embedding creation, Retrieval-Augmented Generation (RAG) database storage and retrieval, and LLM-based query processing and summarization. Additionally, the system will feature synchronous APIs for handling error reporting and metrics evaluation. Real-time monitoring and logging will be enabled through the ELK stack, ensuring efficient service communication, error tracking, and overall system performance.

1. Interfaces (Request/Response)

- **API Gateway**: Utilize **AWS API Gateway** or **Google Cloud Endpoints** to expose RESTful APIs that handle status checks, user queries, and results retrieval. This will act as the central point for managing client requests and routing them to the appropriate backend services.

Input Interface:

The API will accept documents in various formats (e.g., DOC/DOCX, PDF, TXT, JSON, JPEG, PNG, etc.). In addition, metadata or document processing instructions will be submitted in **JSON** format. The API will also accept free-text user queries related to the documents, enabling a flexible input structure.

Output Interface:

The API will return a summary of the processed document in a structured **JSON** format, including:

- Extracted text after chunking (processed content).
- Embedding vectors for each chunk.
- Any errors encountered during processing (e.g., invalid document format, missing required fields).
- Results in response to the free-text user queries, based on the document content.
- Metrics of system results in response to the free-text user queries

2. Document Ingestion and Storage Service

- **Document Reception**: The system will support multiple file formats such as DOC/DOCX, PDF, TXT, JSON, JPEG, PNG, and JIF.
- **Upload & Storage**: Documents will be uploaded via API endpoints using services like **Boto3** for AWS or **Google Cloud Storage API** for GCP. These files will be stored in object storage services like **AWS S3** or **GCP Cloud Storage**, enabling easy access and retrieval by the processing pipeline.

- **File Validation**: Ensure that received files conform to the required specifications, such as file type and metadata. This validation can be performed using **AWS Lambda** functions or **Google Cloud Functions**. After successful validation, documents will be stored in the respective cloud storage service for further processing.
- **Synchronous API Service**: Expose a REST API service to handle document error reports. This service will send structured **JSON** containing details about document errors and failures.

3. Document Parsing Service

- **Document Parsing**: The system will utilize specialized tools for parsing various document types:
    - **PDF Parsing**: Use tools like **Apache Tika**, **PyMuPDF**, or **PDFplumber** to extract text from PDFs.
    - **Text Extraction**: For DOC/DOCX files, **python-docx** will be used. For image formats (JPEG/PNG), **Optical Character Recognition (OCR)** will be performed using services like **AWS Textract** or **Google Cloud Vision**.
- **Synchronous API Service**: Expose a REST API service to handle document error reports. This service will send structured **JSON** with details about document errors and failures encountered during the parsing process.

4. RAG Insert Service

- **Natural Language Processing (NLP)**: After text extraction, NLP models and techniques (e.g., entity recognition, relation extraction, and free-text analysis) will be used to process the extracted data.
- **Text Chunking**: The extracted text will be split into smaller chunks using tools like **RecursiveCharacterTextSplitter** or other **Langchain** alternatives.
- **Embedding Creation**: Leverage pre-trained models like **sentence-transformers-alephbert** to generate embeddings for Hebrew text.
- **Insert into RAG DB**: The embedding vectors will be inserted into a **RAG** (Retrieval-Augmented Generation) database, such as a **Postgres vector addon** (pgvector) or similar solution, for efficient querying and storage.

5. LLM Response & RAG Retrieval Service

- **Synchronous API Service**: A REST API service will handle free-text user queries, receive structured **JSON**, and return the **LLM**-generated responses in **JSON** format.
- **Embedding Creation**: Use pre-trained models like **sentence-transformers-alephbert** to generate embedding vectors for the user queries.
- **Querying the RAG DB**: The service will query the RAG database and return the **k most relevant chunks** based on the similarity threshold.
- **Prompt Creation**: A prompt will be generated using the context of the user query and the most relevant chunks retrieved from the RAG DB.
- **Summarization Model**: Use a model like **dictalm2.0-instruct:f16** to generate an automatic summary based on the context extracted from the document.

- **Summary Creation**: The summarization model will condense the key points of the document into a meaningful and concise summary.

6. Metrics and Evaluation

- **Synchronous API Service**: A REST API service will handle metric requests for free-text user query responses. The service will return **JSON** format responses containing evaluation metrics for LLM responses.
- **BERT Similarity Evaluation**: Use **transformers** and **bert_score** libraries to calculate the similarity between the LLM's response and the correct answers using evaluation metrics like **Precision**, **Recall**, **F1**, and **Similarity**.

7. Monitoring, Logging, and Service Communication

- **Error Monitoring and Logging**: All system services (Document Ingestion, Parsing, RAG Insert, LLM Response, etc.) will integrate with an **ELK stack** (Elasticsearch, Logstash, Kibana) for centralized logging and monitoring. This will enable:

    - **Log aggregation** across all components, providing a single view of the system's operational state.
    - **Error tracking** to identify and resolve failures or performance bottlenecks in real-time.
    - **Visualization and alerts** using **Kibana** dashboards for monitoring the health of the system and identifying potential issues.
- **Service Communication**: Services will communicate with each other via REST APIs, ensuring modularity and scalability. Each service will send detailed logs, performance metrics, and error reports to the **ELK stack**, allowing for end-to-end visibility and the ability to diagnose issues quickly.

- **Error Handling and Notifications**: The system will have robust error-handling mechanisms, including retries and fallback procedures, for document uploads, parsing, and querying. Any errors that cannot be resolved immediately will trigger alerts in the monitoring system, providing real-time notification to the development and operations teams.